Intel® Graphics Media Accelerator 900 Series Software Developer's Guide

Version 2.0

by Adam Lake

February 23, 2005

The information contained in this document is provided for informational purposes only and represents the current view of Intel Corporation Intel on the date of publication. Intel makes no commitment to update the information contained in this document, and Intel reserves the right to make changes at any time, without notice.

DISCLAIMER. THIS DOCUMENT AND ALL INFORMATION CONTAINED HEREIN IS PROVIDED AS IS. INTEL MAKES NO REPRESENTATIONS OF ANY KIND WITH RESPECT TO PRODUCTS REFERENCED HEREIN, WHETHER SUCH PRODUCTS ARE THOSE OF INTEL OR THIRD PARTIES. INTEL EXPRESSLY DISCLAIMS ANY AND ALL WARRANTIES, IMPLIED OR EXPRESS, INCLUDING WITHOUT LIMITATION, ANY WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, NON-INFRINGEMENT, AND ANY WARRANTY ARISING OUT OF THE INFORMATION CONTAINED HEREIN, INCLUDING WITHOUT LIMITATION, ANY PRODUCTS, SPECIFICATIONS, OR OTHER MATERIALS REFERENCED HEREIN. INTEL DOES NOT WARRANT THAT THIS DOCUMENT OR THE INFORMATION CONTAINED HEREIN IS FREE FROM ERRORS, OR THAT ANY PRODUCTS OR OTHER TECHNOLOGY DEVELOPED IN CONFORMANCE WITH THIS DOCUMENT WILL PERFORM IN THE INTENDED MANNER, OR WILL BE FREE FROM INFRINGEMENT OF THIRD PARTY PROPRIETARY RIGHTS, AND INTEL DISCLAIMS ALL LIABILITY THEREFOR.

INTEL DOES NOT WARRANT THAT ANY PRODUCT REFERENCED HEREIN OR ANY PRODUCT OR TECHNOLOGY DEVELOPED IN RELIANCE UPON THIS DOCUMENT, IN WHOLE OR IN PART, WILL BE SUFFICIENT, ACCURATE, RELIABLE, COMPLETE, FREE FROM DEFECTS OR SAFE FOR ITS INTENDED PURPOSE, AND HEREBY DISCLAIMS ALL LIABILITIES THEREFOR. ANY PERSON MAKING, USING OR SELLING SUCH PRODUCT OR TECHNOLOGY DOES SO AT HIS OR HER OWN RISK.

Licenses may be required. Intel and others may have patents or pending patent applications, trademarks, copyrights or other intellectual proprietary rights covering subject matter contained or described in this document. No license, express, implied, by estoppels or otherwise, to any intellectual property rights of Intel or any other party is granted herein. It is your responsibility to seek licenses for such intellectual property rights from Intel and others where appropriate.

Limited License Grant. Intel hereby grants you a limited copyright license to copy this document for your use and internal distribution only. You may not distribute this document externally, in whole or in part, to any other person or entity.

LIMITED LIABILITY. IN NO EVENT SHALL INTEL HAVE ANY LIABILITY TO YOU OR TO ANY OTHER THIRD PARTY, FOR ANY LOST PROFITS, LOST DATA, LOSS OF USE OR COSTS OF PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES, OR FOR ANY DIRECT, INDIRECT, SPECIAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF YOUR USE OF THIS DOCUMENT OR RELIANCE UPON THE INFORMATION CONTAINED HEREIN, UNDER ANY CAUSE OF ACTION OR THEORY OF LIABILITY, AND IRRESPECTIVE OF WHETHER INTEL HAS ADVANCE NOTICE OF THE POSSIBILITY OF SUCH DAMAGES. THESE LIMITATIONS SHALL APPLY NOTWITHSTANDING THE FAILURE OF THE ESSENTIAL PURPOSE OF ANY LIMITED REMEDY.

Intel, the Intel logo, Pentium, Intel Xeon, and VTune are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries.

*Other names and brands may be claimed as the property of others.

Copyright © 2005 Intel Corporation

Table of Contents

1 Introduction

Intel® Integrated Graphics provides a compelling choice for software developers deciding upon graphics platforms for which to develop, test, and optimize. This solution enjoys widespread deployment on mainstream personal computers, as many consumers have found that it provides the performance they need at low cost. Thus, it pays to keep in mind the experience that these customers will have when they buy your game, and to take steps to ensure that their experience will be a good one. This document describes the Intel® Graphics Media Accelerator (Intel® GMA) and provides key development hints and tips that will help ensure that your customers have a great time playing your games.

Intel Graphics is designed to meet the display needs for the majority of business and consumer users who do not require expensive 3D processing. The graphics core is built into the chipset and integrated into the motherboard. It utilizes the central processing unit (CPU) to assist in rendering 3D tasks and shares system memory with the operating system to significantly lower the system cost. As Intel's latest graphics core, the Intel GMA is integrated into the Intel® 915G Express Chipset family and provides an very high degree of graphics performance for very little system cost. Intel GMA's capabilities match or exceed many more expensive graphics card solutions. PC buyers have appreciated this balanced approach to system design, and Intel Graphics is currently the number one graphics solution chosen by new PC purchasers[1]

2 Features of the Intel GMA Graphics Core

This section describes the features of the Intel GMA graphics core and describes in detail the architectural differences in each of the recent generations of Intel GMA graphics. It also gives a comparison to some discrete card solutions to provide an idea of the performance you can expect from Intel Integrated Graphics.

2.1 Highlights of the Intel GMA Graphics Core

- 256-bit graphics core running at 400MHz
- Up to 10.6GB/sec memory bandwidth with DDR2 667 system memory
- 1.6GPixels/sec and 1.6GTexels/sec fill rate
- 224MB maximum video memory
- 2048x1536 at 75Hz maximum resolution
- Dynamic Display Modes for flat-panel, wide-screen, and digital TV support
- Operating systems supported: Microsoft Windows* XP, Windows XP 64-bit, Media Center Edition 2004/2005, Windows 2000; Linux*-compatible (Xfree86 source available)

2.2 High Performance 3D

- Up to four pixels per clock rendering
- Microsoft DirectX* 9 hardware acceleration features

- Pixel Shader* 2.0
- Volumetric Textures
- Shadow Maps
- Slope Scale Depth Bias
- Two-Sided Stencil
- Microsoft DirectX 9 Vertex Shader 2.0 and Transform and Lighting supported in software through highly optimized Processor Specific Geometry Pipeline (PSGP)
- Texture Decompression for DirectX and OpenGL*
- OpenGL 1.4 support plus ARB_vertex_buffer, EXT_shadow_funcs extensions, ARB_vertex_program and ARB_fragment_program

2.3 Advanced Display Technology

- Up to 2048x1526 resolution for both analog and digital displays
- Consumer electronic display (digital TV) support
- Display hot plug support to automatically detect new display connection while system is operating (CRT and DVI)
- Two Serial Digital Video Out (SDVO) ports for flat-panel monitors and/or TV-out support via Advanced Digital Display 2 (ADD2) cards
- ADD2+ providing ADD2 and PCI Express* x1 on a single adapter in the PCI Express x16 port
- TV-out and PVR capability with ADD2+ cards
- Multiple display types (LVDS, DVI-I, DVI-D, HDTV, TV-out, CRT) for dual-monitor capabilities
- HDTV 720p and 1080i display resolution support
- Interlaced Display output support
- 16x9 and 16x10 Aspect Ratio for widescreen displays
- 2x2 Panel Scaler

2.4 High Quality Media Support

- Up and down scaling of video content
- High Definition Content Decode – up to two stream support
- 5x3 Overlay Filtering
- Hardware Motion Compensation support through four pipes

2.5 Feature Comparison to Previous Intel Graphics Generations

Feature/Specs	Intel® 845G Chipset Intel® Extreme Graphics	Intel® 865G Chipset Intel Extreme Graphics 2	Intel® 915G Chipset Intel GMA 900
DirectX Hardware Acceleration	DirectX 7.1	DirectX 7.1	DirectX 9
OpenGL Version Support	1.3	1.3	1.4
Core Clock	200MHz	266MHz	333MHz
Pixel Pipes	1	1	4
Display Support	Single (dual clone only)	Single (dual clone only)	Dual Independent
Total Video Memory (max dynamic)	64MB	96MB	224MB
Fixed Video Memory (max fixed)	8MB	16MB	128MB1
Memory Bandwidth (Max config)	2.1GB/s	6.4GB/s	8.5GB/s
Display Support	RGB	RGB (QXGA) DVI, Composite, S-Video (via DVO)	RGB (QXGA) DVI, HDTV (1080i, 720p), Composite, Component, S-Video (via Intel® SDVO)
Max Resolution	QXGA 60Hz (2048x1536)	QXGA 60Hz (2048x1536)	QXGA 75Hz (2048x1536)

2.6 Feature Comparison to Comparable Discrete Adapters

	Intel® 915G Chipset	FX5200 (nVidia*)	9200 (ATI*)
Graphics Name	Intel GMA 900 Graphics	NV34	RV250
Timeframe (MP)	Q2 '04	Q2'03	Q2'03
Graphics Core Speed	333MHz	250MHz	250MHz
Memory Support	Up to DDR2 533	DDR 333	DDR2 500
Memory BW	Up to 8.5GB/s	5.3GB/s	4GB/s
Fill Rate	1.3GT/s	1GT/s	1GT/s
Pixel Rate	1.3GP/s	1GP/s	1GP/s
Pixel Pipes	4	4	4
Pixel Shader	2.0	2.0	1.4
Vertex Shader	3.0 (SW)	2.0	1.1
Direct X HW Support	9.0	9.0	8.1
Dual Independent Display	Yes (available option)	Yes	Yes
Digital Display Support	TV/DVI (available option)	TV/DVI	TV/DVI

3 Architecture Considerations for the Programmer

3.1 Zone Rendering

Zone Rendering is designed to reduce memory bandwidth and to maximize rendering performance by rendering the frame buffer as a set of tiles. To understand how it works, we must first have an understanding of conventional rendering. With conventional rendering, a scene made up of various 3D models is sent to the graphics hardware, where each model and its associated polygons undergo a series of matrix multiplications to transform them from model space (the local coordinate system of each object) to world space (the coordinate system relative to the entire scene) and finally to view space (the viewer's coordinate system). Next, light values are applied to the vertices of each triangle, which are then converted into pixel or screen coordinates. Next, we rasterize the polygons via bilinear interpolation. Each pixel is depth tested (depth testing is also known as z-testing, as **z** represents the depth from the screen directly back in towards the monitor) and then given the proper color if it is determined to be nearer than the previously rendered object that occupies that region of space. In the case of the 915G, polygons are received from the Processor Specific Graphics Pipeline (PSGP) running on the CPU.

Zone Rendering aims to improve memory efficiency by reducing memory traffic. As in conventional rendering, the scene is passed to hardware, where the polygons of which it consists are transformed into view space and their vertices lit. Rather than going directly through screen space conversion, however, Zone Rendering first sorts each polygon by

zone. Since each zone can fit in the chipset's on-chip cache, the depth-testing and pixel-blending operations are done quickly on-chip. This also means that each pixel is written to frame-buffer memory only once, reducing memory traffic for a scene.

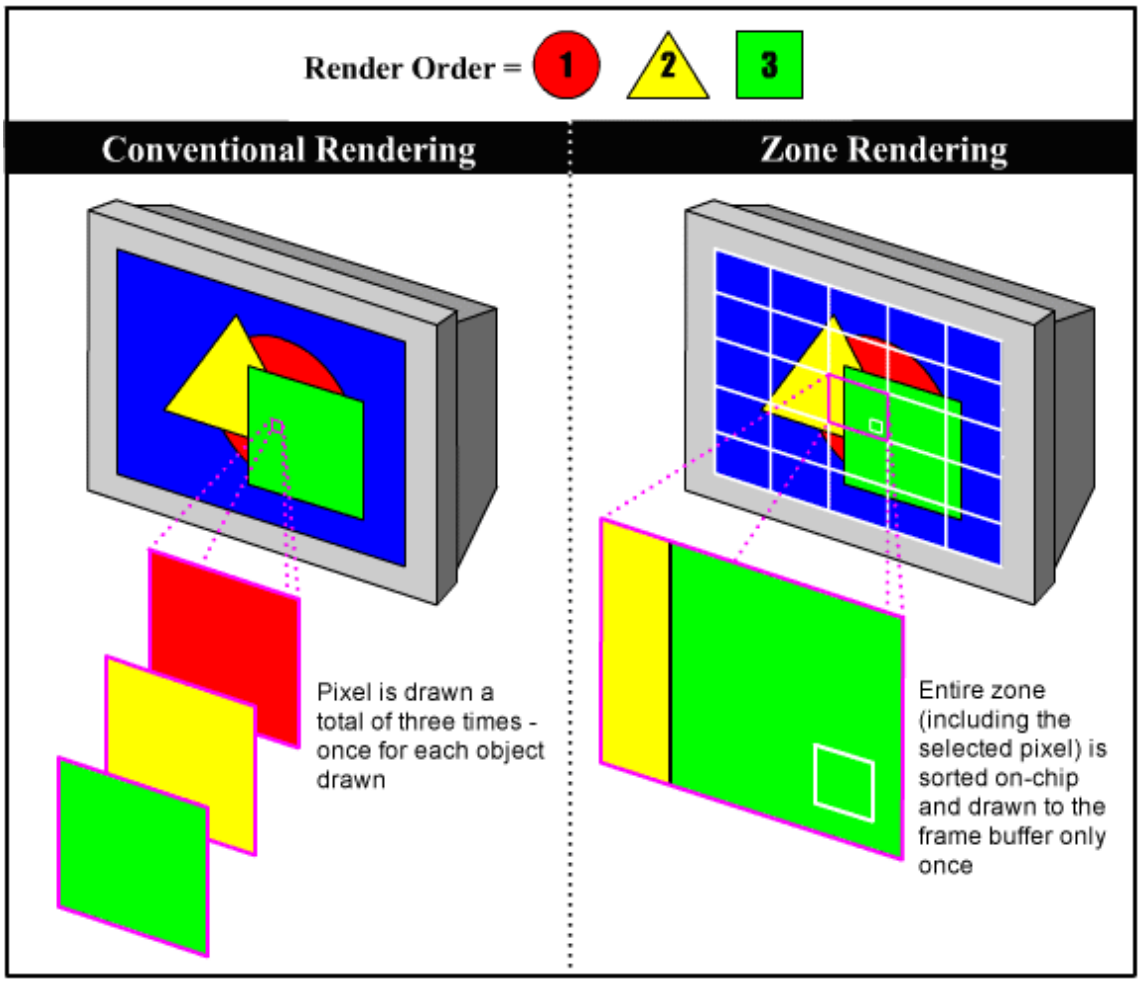


Figure 1. Conventional Rendering versus Zone Rendering.

The amount of memory bandwidth required to render a scene with conventional rendering can be significantly more than the amount required to render a scene with Zone Rendering. The impact to memory bandwidth (and performance) of using Zone Rendering is directly related to the depth complexity of a given scene. While there are no specific coding techniques required to enable Zone Rendering, there are several things a programmer can do to ensure that they take advantage of the performance benefits Zone Rendering has to offer. These tips are detailed in Section 5 of this document, "Tips for Maintaining Zone Rendering."

3.2 Dynamic Video Memory

Intel Graphics utilizes a shared memory architecture (often referred to as a unified memory architecture or UMA) – system memory is used for both graphics and system purposes. Instead of using dedicated local memory, as is the case on the majority of discrete graphics cards, a portion of the system memory is allocated to be used as video memory. Additionally, a small amount of system memory is permanently allocated to video memory by the BIOS. This amount is usually one or eight megabytes.

Dynamic Video Memory Technology (DVMT) allows additional system memory to be dynamically allocated for graphics usage based on application need. Once the application is closed, the memory that was allocated is released and is then available for system use. The purpose of dynamically allocating memory for graphics use is to ensure a solid balance between system performance and graphics performance.

For example, if a user is editing text, there is no need for the graphics to take up a large amount of the system's memory. In such a case, it would be best if more memory were allocated to the system. On the other hand, if the user were to start up a 3D game, there would be a need for more of the shared memory to be used as graphics memory.

On boot-up, the user can choose in the system's BIOS the amount of system memory to be used permanently by the graphics controller. Once selected, this memory will never be given back to the system. With drivers older than today's PV14.x Intel Graphics drivers, this memory is also reported as local video memory in DirectX applications. Once the operating system is started, the graphics driver will dynamically allocate graphics memory based on requests from each application run by the user. For systems with 128MB or less of system memory, a *maximum* of 64MB will be set aside for graphics (memory set aside by the BIOS, plus memory dynamically allocated by the driver). For systems with 256MB memory or more, a *maximum* of 128MB will be allocated for use by the graphics controller. In addition, a new BIOS setting can provide 128MB of fixed memory or even 224MB of memory for systems with 512MB or more memory.

The following table compares the maximum video/graphics memory that can be allocated (based on total system memory) for the Intel® 845G, 865G, 915G, 915GM, and 945G chipsets:

System Memory	• 64MB	• 128MB	• 256MB	• 512MB
Chipset	**Maximum Graphics Memory**			
Intel 845G	32MB	64MB	96MB	96MB
Intel 865G	32MB	64MB	96MB	96MB
Intel 915G	32MB	64MB	128MB	224MB*
Intel 915GM	32MB	64MB	128MB	224MB*
Intel 945G	32MB	64MB	128MB	224MB*

* When MAX DVMT is set within the system's BIOS.

3.3 Software Vertex Processing

As discussed earlier, Intel Integrated Graphics relies on the CPU to perform transformation and lighting of the vertices. Thus, all vertex shader operations are executed on the CPU. This portion of the graphics pipeline is known as the *Processor Specific Geometry Pipeline* (PSGP)[2]. Several things are done in the PSGP to improve performance. First, it is a SIMD-optimized transform and lighting pipeline tuned specifically for vertex processing in today's gaming applications. Second, input streams are prefetched to hide memory latency. If no readback is required and the application programmer has made DirectX aware of this, the vertex buffers are processed and written to driver memory. If the same vertex buffer is processed in both software and

hardware, it is advisable to create multiple copies of the vertex buffer and place one in system memory and one in driver memory.

3.3.1 Reduce Clipping When Using Software Vertex Processing

It also helps increase performance to use the *D3DUSAGE_DONOTCLIP* flag if you know the primitive will never need to be clipped. When clipping is enabled, the PSGP stores its output to system memory instead of being able to write directly to driver-allocated memory. Using bounding volumes to reject objects that will never be in view is a good way to reduce the number of primitives clipped. Additionally, objects that are contained entirely in the viewport can also be specified as *D3DUSAGE_DONOTCLIP*. The PSGP also performs primitive batching of the processed vertices before sending them to hardware, thus reducing the number of hardware vertex buffer changes. As long as the VB output FVF matches for a given set of vertex buffers, they can be batched together. Unlike hardware vertex processing, changing the VB FVF does not create a slowdown.

3.3.2 Vertex Shaders and Software Vertex Processing

At vertex shader creation, the shader code is compiled to the equivalent IA-32 code using all available assembly optimizations and instructions on the client's CPU to achieve the fastest code possible. At vertex shader execution, the already generated vertex shader is called. In general, software vertex shaders have very good performance. One reason for this high performance is that the frequency of the CPU is much higher frequency than today's 3D graphics chips; therefore, the number of instructions retired per unit of time can be much higher. With that said, it is important to understand that one shader instruction may not take one clock tick to execute for both hardware and software vertex shaders. See Section 7, "Tools to Help in Programming the 915G," for more information on optimizing vertex shaders for Intel Integrated Graphics.

4 Programming for the 915G

4.1 Identifying the Intel 915G

To target features specifically on Intel Graphics, Device and Vendor ID information should be used to properly detect the device. This can be read from the PCI configuration space or through DirectX (version 8 or later) by using the `GetAdapterIdentifier()` function.

The following code snippet shows how to identify Intel Graphics:

```
DWORD behaviorFlags;              // Used to describe vertex processing
type
D3DADAPTER_IDENTIFIER9 adapterID;  // Used to store device info


// Gather the primary adapter's information...
if(g_pd3dDevice->GetAdapterIdentifier( 0, 0, &adapterID ) != D3D_OK )
   exit(-1);
```

```
if ( ( adapterID.VendorId == 0x8086 ) &&
   ( ( adapterID.DeviceId == 0x2772 ) ||  //Intel 945G Controller 0
     ( adapterID.DeviceId == 0x2776 ) ||  //Intel 945G Controller 1
     ( adapterID.DeviceId == 0x2592 ) ||  //Intel 915GM Controller 0
     ( adapterID.DeviceId == 0x2792 ) ||  //Intel 915GM Controller 1
     ( adapterID.DeviceId == 0x2582 ) ||  //Intel 915G Controller 0
     ( adapterID.DeviceId == 0x2782 ) ||  //Intel 915G Controller 1
{
   // An Intel graphics chipset is active...
   .
   .
   .
}
```

4.2 Identifying Software TNL

Use the Direct3D* *D3DCREATE_SOFTWARE_VERTEXPROCESSING* definition to create a device that uses software vertex processing. The following sample DirectX 9 function detects Intel Graphics and enables Software Vertex Processing (see Appendix for full source):

```
//-----------------------------------------------------------------------
---
// Name: SetVertexProcessingMode
// Desc: Checks HW TnL caps and IDs Intel Graphics to enable SW TnL
//-----------------------------------------------------------------------
---
DWORD SetVertexProcessingMode( LPDIRECT3D9 pD3D )
{
    DWORD     vertexprocessingmode;  // vertex processing mode
    D3DCAPS9  caps;                  // structure that stores device
caps...
    D3DADAPTER_IDENTIFIER9 adapterID;  // Used to store device info


    // Check the capabilities...store into "caps"...
    if( g_pD3D->GetDeviceCaps( 0, D3DDEVTYPE_HAL, &caps ) != D3D_OK )
    {
   return E_FAIL; // exit if reading caps fails...
    }
    // check if hardware TnL is supported...
    if ( ( caps.DevCaps & D3DDEVCAPS_HWTRANSFORMANDLIGHT ) != 0 )
    {
```

```
   vertexprocessingmode = D3DCREATE_HARDWARE_VERTEXPROCESSING;
    }
    else
    {
       // check vendor and device ID and enable software vertex
processing
       // for Intel(R) Graphics
       // Gather the primary adapter's information...
   if( g_pD3D->GetAdapterIdentifier( 0, 0, &adapterID ) != D3D_OK )
   {
      return E_FAIL;
   }


if ( ( adapterID.VendorId == 0x8086 ) &&
   ( ( adapterID.DeviceId == 0x2772 ) ||  //Intel 945G Controller 0
     ( adapterID.DeviceId == 0x2776 ) ||  //Intel 945G Controller 1
     ( adapterID.DeviceId == 0x2592 ) ||  //Intel 915GM Controller 0
     ( adapterID.DeviceId == 0x2792 ) ||  //Intel 915GM Controller 1
     ( adapterID.DeviceId == 0x2582 ) ||  //Intel 915G Controller 0
     ( adapterID.DeviceId == 0x2782 ) ||  //Intel 915G Controller 1
                       {
           vertexprocessingmode = D3DCREATE_SOFTWARE_VERTEXPROCESSING;
   }
   else
   {
      // chip does not meet requirements...
      return E_MINSPEC;
   }
   }
   return vertexprocessingmode;
}
```

4.3 Rethinking Assumptions about Hardware TNL

A common assumption made by application developers is that hardware transformation and lighting is required to achieve a certain level of performance – the intended result is a “good experience” or no experience at all. However, Intel Graphics provides a

balanced graphics pipeline by allowing the Pentium® 4 processor to perform the transformation and lighting operations, which is often more than capable of achieving a good experience.

In using the CPU for transform and lighting operations, the DirectX transform and lighting pipe optimized for the Pentium 4 processor is utilized. This PSGP allows the Intel® Extreme Graphics controller to offload the transform and lighting operations to software while still providing excellent performance.

If application performance requirements are based on actual performance, rather than on assumptions of performance based on features, this should allow for a good experience on many systems.

4.4 Device Capabilities

Listing all of the device capabilities supported by any piece of graphics hardware is a very large undertaking. The easiest way to examine a particular graphics device capability is to use the DirectX Caps Viewer, available after the DX SDK is installed on a system. This provides detailed information for each capability of the graphics card when running a DirectX driver. When using this mode, be sure to select the appropriate adapter format that represents the mode in which the graphics driver is running. For example, *D3DFMT_X8R8G8B8*, full screen, windowed, and so on.

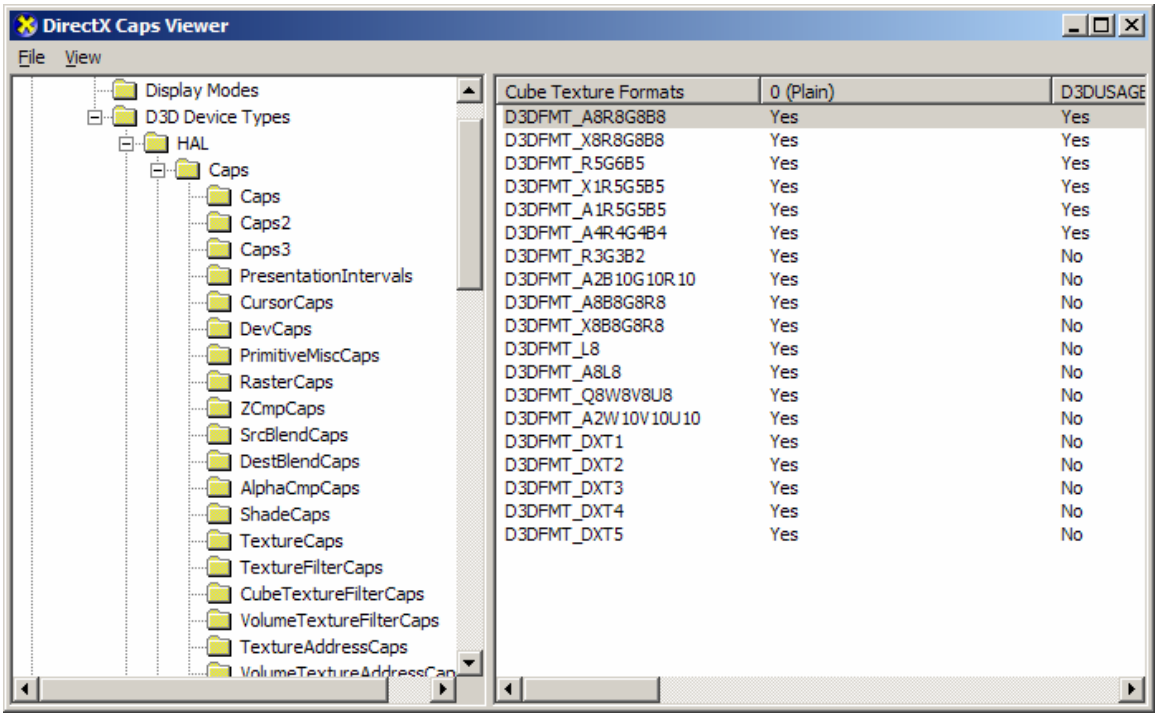


Figure 2. A snapshot of the DirectX Caps viewer. On the left is a list of capability topics. On the right are the detailed caps bits reported by this hardware.

4.5 Checking Device Capabilities within Your Application

It has become increasingly common to check for certain device capabilities (such as the number of vertex streams a device is capable of supporting or the maximum number of lights that can be made active) before allowing a game or application to run. Intel Graphics takes advantage of the processor for some work, and because of that fact, it can actually provide ambiguous capability reports on some older drivers.

DirectX allows you to check the capabilities of a device using either `IDirect3DDevice9::GetDeviceCaps()` or `IDirect3D9::GetDeviceCaps()`. When checking for anything outside of the vertex processing support capabilities of a device, use `IDirect3DDevice9::GetDeviceCaps()`, since an IDirect3DDevice9 object requires the vertex processing mode to be set. In the case of Intel Graphics, this needs to be created using *D3DCREATE_SOFTWARE_VERTEXPROCESSING*. This setting tells the Intel Graphics driver that the device will be taking advantage of the PSGP and allows the driver to adjust what it reports out to match those capabilities supported by the PSGP. The table below shows how the two methods report differently on a sampling of device capabilities:

Capability	IDirect3D9::GetDeviceCaps()	IDirect3DDevice9::GetDeviceCaps()
MaxActiveLights	0	0xffffffff
MaxUserClipPlanes	0	6
MaxVertexBlendMatrices	0	4
MaxStreams	1	16
VertexShaderVersion	0.0	3.0

4.6 Checking for Available Memory

A check that is often performed before actually executing the application checks the amount of available free graphics or video memory. As a result of the dynamic allocation of graphics memory performed by the Intel Integrated Graphics devices (based on application requests), you need to take care in ensuring that you understand all of the memory that is truly available to the graphics device. Memory checks that only supply the amount of 'local' graphics memory available do not supply an appropriate value for the Intel Integrated Graphics devices.

To accurately detect the amount of memory available to the Intel Integrated Graphics devices, check the total video memory availability. Local memory is considered as the memory permanently set aside by the BIOS for use as graphics memory. Non-local video memory is the memory beyond the local video memory that was dynamically allocated based on requests from applications. Both local and non-local video memory combine to equal the total amount of video memory, and each is handled identically for memory accesses.

The code snippet below outlines the function calls necessary to most accurately check the memory available for use by the graphics controller within DirectX 7:

```
...

DDSCAPS2 ddsVidMemcaps;
ZeroMemory(&ddsVidMemcaps, sizeof(DDSCAPS2));
ddsVidMemcaps.dwCaps = DDSCAPS_VIDEOMEMORY;

hRet = g_pDD->GetAvailableVidMem(&ddsVidMemcaps, &dwVidMemTotal,
       &dwVidMemFree);

...
```

The corresponding solution in DirectX 8 and 9 is shown below:

```
...
```

```
int AvailableTextureMem = g_pd3dDevice->GetAvailableTextureMem();

...
```

4.7 Depth Texture Support

In OpenGL, Intel GMA Graphics is supported using the depth buffer as a texture. For more information on Depth Texture support, see the documentation for ARB_depth_texture*. In Direct3D, depth textures are not supported.

4.8 Floating Point Textures and Render Targets

The Intel GMA 900 series does not support floating-point texture formats as render targets in Direct3D or OpenGL. For applications that need or want to use floating-point textures, for example for high dynamic range images, there are other ways to simulate floating-point support using integer textures. Refer to Adam Lake and Cody Northrup's High Dynamic Range Environment Mapping on mainstream graphics hardware, listed in the Additional Resources section of this document, for more details. To determine if your particular texture format is supported see Section 4.5, "Checking Device Capabilities within Your Application."

4.9 Support for Queries via the DirectX CreateQuery(...) Interface

Intel GMA 900 series does not currently support the queries that can be issued with the `CreateQuery(…)` interface.

4.10 Vertex Shader and Pixel Shader Support

The Intel GMA 900 Series supports Pixel Shader 2.0 and Vertex Shader 3.0. For hardware shaders written in another version of vertex or pixel shaders, the shader is translated to an equivalent but optimized vs 3.0/ps 2.0 code path. Compile and author shaders in Vertex Shader 3.0 and Pixel Shader 2.0 for best performance. Additionally, there is a small impact on performance when the shaders have to be translated. Fortunately, this only has to happen when the shader is first compiled. Section 6, "Programmable Graphics Pipeline," has more details on vertex and pixel shader optimization for the Intel GMA 900 Series.

4.11 Lack of Native Anti-Aliasing Support

We do not support anti-aliasing in OpenGL or DirectD3D in hardware. If you require anti-aliasing, you can do this in your engine by rendering at a higher resolution, then down-sampling into the frame buffer. Be aware that this will have a dramatic impact on performance.

4.12 Texture Filtering

Under DirectX, all forms of texture filtering are supported. These include the following:

- `D3DTEXF_NONE`
- `D3DTEXF_POINT`
- `D3DTEXF_LINEAR`
- `D3DTEXF_ANISOTROPIC`
- `D3DTEXF_PYRAMIDALQUAD`
- `D3DTEXF_GAUSSIANQUAD`

In OpenGL, the following texture filtering operations are supported:

- `GL_NEAREST`
- `GL_LINEAR`
- `GL_NEAREST_MIPMAP_NEAREST`
- `GL_LINEAR_MIPMAP_NEAREST`
- `GL_LINEAR_MIPMAP_LINEAR`
- anisotropic filtering via *EXT_texture_filter_anisotropic*

5 Tips for Maintaining Zone Rendering

Enabling Zone Rendering technology can result in a significant increase in system performance. However, in order to maximize the benefit of Zone Rendering on a given application, there are some recommendations of which you need to be aware. In some cases, the integrated graphics device does not have the resources available to efficiently render by zone. In these cases, the Intel Integrated Graphics devices are forced out of Zone Rendering mode into a classic rendering mode. Intel recommends that you attempt to maintain Zone Rendering whenever possible. While it is difficult to know whether Zone Rendering is enabled, there are several things you should check if you believe that Zone Rendering is not enabled (e.g., if you see a lower than expected frame rate). The topics discussed below represent the most common reasons for the Intel Integrated Graphics devices to be forced out of Zone Rendering mode into classic rendering mode.

5.1 Change Render Targets Outside of Scene

Changing rendering targets is used for creating a variety of effects. Where these calls are made can have a significant performance impact on the performance of Intel Extreme Graphics 2. If any changes to the render target are made, they should be done before or after the scene is rendered. For example, if a render to texture technique is used to create a shadow effect, the render to texture could be completed at a variety of points. Completing the render to texture before the scene will result in increased performance by allowing Zone Rendering mode to stay active and by avoiding unnecessary buffer evictions.

5.2 Avoid Reading Back from Buffers

Reading back from color, z, or stencil buffers has a significant impact to performance on any graphics chip. The read-backs themselves are slow and can potentially cause the Intel Integrated Graphics devices to be forced out of Zone Rendering mode into classic rendering mode, which further degrades performance. Intel recommends avoiding reading back data from any of the buffers, but the z-buffer in particular is most costly on Intel Graphics and perhaps the most avoidable. One advantage of Zone Rendering is that a z-buffer does not need to be maintained at all. Reading back from the z-buffer, however, forces the maintenance of a z-buffer and will degrade performance.

5.3 Avoid Locking the Buffers

Locking buffers is often done for synchronization reasons or occasionally for creating visual effects. Unfortunately, locking buffers can have a significant impact to performance and will possibly cause the Intel Integrated Graphics devices to be forced out of Zone Rendering mode into classic rendering mode. Therefore, lock buffers sparingly, if at all.

5.4 Buffer Clears

There are many reasons for clearing any one of the buffers (depth, color, or stencil). How these buffers are cleared will determine whether there is a performance impact in the application. Clearing the buffers together and once per frame is the best option. In cases when this is not possible, the depth and stencil buffers should be cleared together. Similarly, partially clearing buffers can have a negative impact on performance. Fast clear options that allow the entire buffer to clear quickly will not be implemented if a buffer is only partially cleared.

5.5 Reduce Memory Bandwidth If Necessary

Zone Rendering improves performance by reducing memory bandwidth. Intel Integrated Graphics devices, however, can be forced out of Zone Rendering mode into classic rendering mode by providing too much geometry, texture, and/or state information to memory. Below is a list of potential ways to reduce the memory bandwidth required to render a scene:

- Use compressed textures.
- Use D3DPOOL_MANAGED or D3DPOOL_DEFAULT when allocating surface, buffer, or texture memory.
- Reduce texture size or quality.
- Use level-of-detail.
- Reduce the content footprint by employing efficient culling algorithms.

6 Programmable Graphics Pipeline

This section contains a detailed description on the advanced features of Intel GMA graphics. The programmable graphics pipeline is one of the most powerful features

found on Intel GMA, and it points the direction for graphics architectures in the near future. It is important, however, to detect these features properly. As stated previously in this guide, be sure to use `IDirect3DDevice9::GetDeviceCaps()` instead of `IDirect3D9::GetDeviceCaps()` to properly detect support for vertex and pixel shading. If you do not do this, your application will detect no vertex shader support at all and revert to a much lower feature set than what is possible on Intel Graphics.

While we hope the information in this section is useful, the articles by Kent F. Knox and Dean Macri identified in the Additional Resources section go into additional detail and are recommended reading for more information on the programmable graphics pipeline and vertex and pixel shaders.

6.1 Software Vertex Shading

In DirectX applications, software vertex shading is done with a highly optimized Streaming SIMD Extensions (SSE) implementation in the PSGP. The PSGP is updated for every new processor launch and is actively maintained by Intel and Microsoft engineers. It automatically takes advantage of the latest instruction sets and is threaded to benefit from Hyper-Threading Technology and multi-core platforms. Vertex Shader versions 2.0, 2.x, and 3.0 are supported.

For OpenGL vertex shading, there is not a PSGP equivalent component. One way to think of it would be the Direct3D (D3D) runtime, PSGP, and D3D driver all rolled into one: the OpenGL driver DLL. The driver uses SSE/SSE2 optimizations for vertex shaders, but it is not currently threaded. IGMA950 supports OpenGL 1.4; the Delphi3D* Web site maintains a full list of OpenGL extension support*.

6.2 Benefits of Software Vertex Shading

There are multiple benefits to using software vertex shading, as opposed to hardware implementations:

Advanced Shaders: Version 3.0 vertex shaders have an unlimited instruction count; therefore you can develop highly advanced shaders or prototype new ideas at will. More recent shader versions allow more complex techniques than those supported by currently available hardware.

Readback Performance: Since the IGMA950 video and system memory are shared, readback performance is not gated by an Accelerated Graphics Port (AGP) or PCI Express (PCIE) interface. This can be beneficial for some algorithms.

Compatibility: Hardware Transform and Lighting implementations are not always guaranteed to behave identically. Using software shading helps to ensure that users will see exactly what you want them to. In addition, when you design shaders to support systems that do not have vertex shaders, your potential audience grows dramatically.

Multiprocessing PSGP: Since dual-core platforms are emerging on industry roadmaps, now is a great time to start taking that advance into account. Of the entire software-graphics pipeline, currently only vertex shading will scale to multiple CPUs. All vertex shader code is just-in-time compiled by D3D for execution on a second thread. When DirectX detects that multiple processors are present, it determines the number of

vertices in each batch, distributes them to multiple processors, and collects the output after all batch processing completes. It would be worthwhile to test your application using PSGP, as multi-core vertex shader animation may be faster than a fixed-function software pipe.

6.3 Tips for Using Software Vertex Shaders

Avoid sparse vertex buffers: Passing a large vertex buffer that only uses a few indices can seriously hinder software vertex shader performance. DirectX processes the entire vertex buffer range of `DrawIndexPrimitive (MinIndex to MinIndex+(NumVertices-1))` before it indexes into the range[3]. This is a different behavior than that seen on discrete cards. It is not safe to presume that you can simply indicate a few indices to draw while passing an enormous vertex buffer to the runtime and assume that the transform will only effect the vertices specified in the `DrawIndexedPrimitive(…)` call. Taking the extra processing effort to minimize your unneeded buffer space can greatly improve your frame rate.

Consistent device settings: It is important to match settings between device and buffer creation, that is, do not create a software device and then try to do hardware vertex processing – your application will crash for the same reasons you cannot use a mixed device. The easiest thing to do here is match your `CreateDevice()` and `CreateVertexBuffer()` behavior flags.

Call `CreateVertexShader()` early: The shader compiler actually creates highly optimized shader code when the application makes this call – do it early so results can be cached.

Geometry instancing with shaders: Multiprocessor optimizations for batched `DrawIndexedPrimitive()` calls allow shader based instancing to perform significantly better than other methods on software devices. See technique #2 of the Instancing Sample in the DirectX 9.0 SDK October 2004 Update* and note the increased framerate versus hardware, constants, or stream instancing.

Force software device during development: Many shader issues can be detected early by forcing your application to run as a software device regularly during development. This will emulate the behavior you will see on software-only systems. In fact, given the market share gains of integrated graphics recently, its becoming more and more likely your application WILL be run on integrated graphics. Thus, making this part of your regular testing procedure will broaden the audience for your game.

6.4 Optimization Tips for Software Vertex Shading

Use the latest shader version: IGMA950 using DirectX 9.0 supports all versions of vertex shaders through Vertex Shader 3.0. You should try to take advantage of all the features provided by the highest shader model to get the best performance possible

Take advantage of macros: Using the macro instructions available in each shader model will greatly reduce instruction counts and allow for greater performance.

Define heavily used constants: Using **def**, **defi**, or **defb** will provide more information to the compiler so it can create the most optimal instruction sequence.

Use source swizzle and destination masks: To minimize SIMD register use, if you do not need the results from an entire vector operation, specify which results you want so the compiler can efficiently allocate resources, as captured in the following table:

Before	After
dp3 r0, v0, r1	dp3 r0.x, v0.xyz, v1.xyz
add r0, r0, v1	add oPos.x, r0.x, v1.x
mov oPos.x, r0.x	

Minimize dependency chains: Try not to structure code such that results from one sequence are required to continue processing. Independent sections of code can be more efficiently scheduled by the compiler.

Use rep instruction if aL is not needed in a loop: Use the **rep** instruction instead of **loop** if the iteration count is known, as this requires less code and fewer registers.

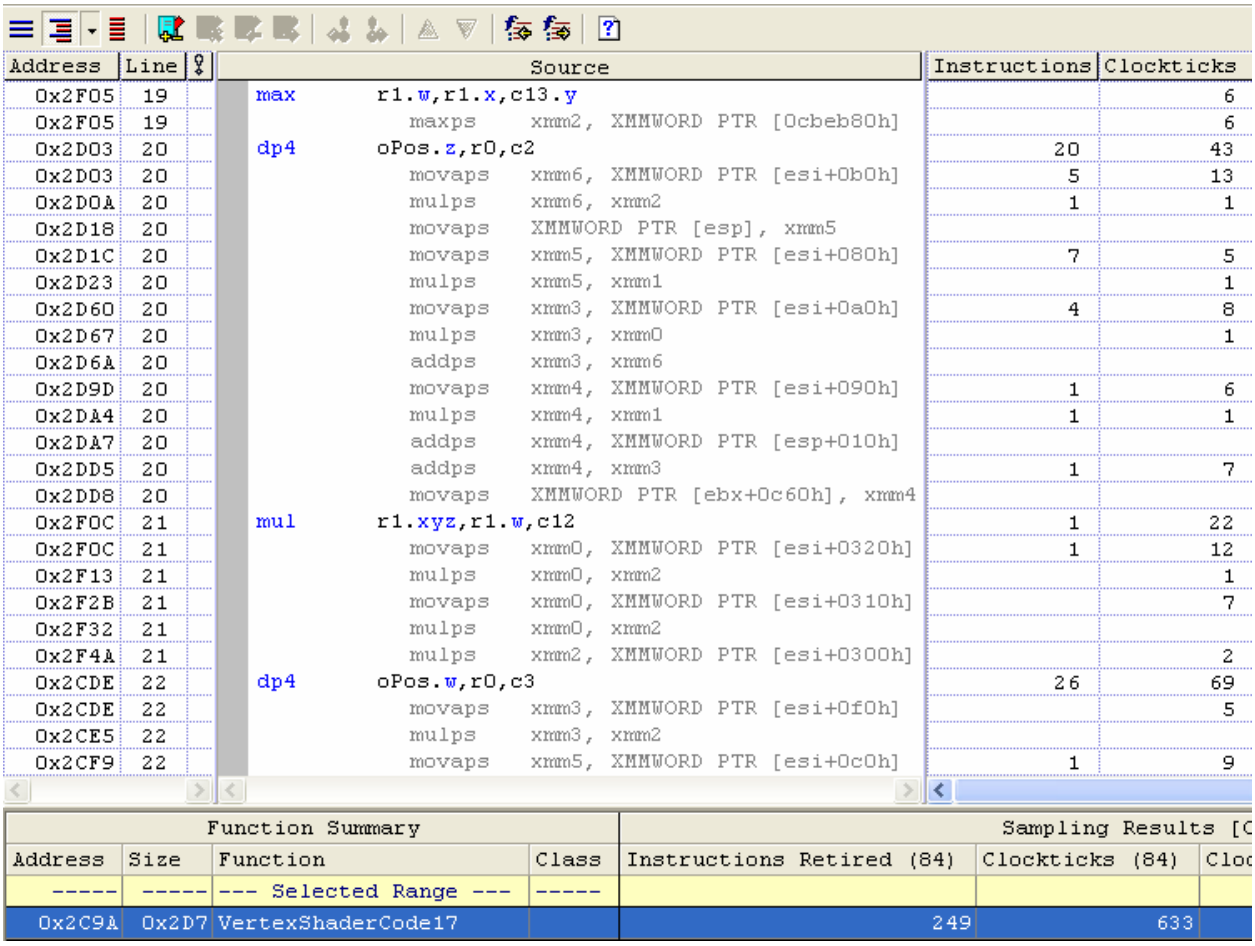
Conditionals: While branching always slows code down, there are a few ways to minimize the impact.

Break when done: Using the break instructions in VS 2.x and 3.0 to early-exit from loops can provide a significant speedup.

Avoid expensive instructions: If your code uses expensive macros like **log** or **pow**, structure your branches to avoid them whenever possible. This will be more efficient than masking.

Intelligent data arrangement: If you know a group of vertices are all going to pass (or fail) a condition, try to execute them together. This will help the compiler generate better branch prediction.

Use the **Intel® VTune™ Performance Analyzer:** In the VTune environment, vertex shaders appear as JIT code. You can drill down into that process to see shader disassembly and potential bottlenecks as displayed in Figure 3 (a trial version of the VTune analyzer is available from the Intel® Evaluation Software Center).



Address	Line	Source	Instructions	Clockticks
0x2F05	19	max r1.w,r1.x,c13.y		6
0x2F05	19	maxps xmm2, XMMWORD PTR [0cbeb80h]		6
0x2D03	20	dp4 oPos.z,r0,c2	20	43
0x2D03	20	movaps xmm6, XMMWORD PTR [esi+0b0h]	5	13
0x2D0A	20	mulps xmm6, xmm2	1	1
0x2D18	20	movaps XMMWORD PTR [esp], xmm5		
0x2D1C	20	movaps xmm5, XMMWORD PTR [esi+080h]	7	5
0x2D23	20	mulps xmm5, xmm1		1
0x2D60	20	movaps xmm3, XMMWORD PTR [esi+0a0h]	4	8
0x2D67	20	mulps xmm3, xmm0		1
0x2D6A	20	addps xmm3, xmm6		
0x2D9D	20	movaps xmm4, XMMWORD PTR [esi+090h]	1	6
0x2DA4	20	mulps xmm4, xmm1	1	1
0x2DA7	20	addps xmm4, XMMWORD PTR [esp+010h]		
0x2DD5	20	addps xmm4, xmm3	1	7
0x2DD8	20	movaps XMMWORD PTR [ebx+0c60h], xmm4		
0x2F0C	21	mul r1.xyz,r1.w,c12	1	22
0x2F0C	21	movaps xmm0, XMMWORD PTR [esi+0320h]	1	12
0x2F13	21	mulps xmm0, xmm2		1
0x2F2B	21	movaps xmm0, XMMWORD PTR [esi+0310h]		7
0x2F32	21	mulps xmm0, xmm2		
0x2F4A	21	mulps xmm2, XMMWORD PTR [esi+0300h]		2
0x2CDE	22	dp4 oPos.w,r0,c3	26	69
0x2CDE	22	movaps xmm3, XMMWORD PTR [esi+0f0h]		5
0x2CE5	22	mulps xmm3, xmm2		
0x2CF9	22	movaps xmm5, XMMWORD PTR [esi+0c0h]	1	9

Function Summary				Sampling Results [C		
Address	Size	Function	Class	Instructions Retired (84)	Clockticks (84)	Cloc
-----	-----	--- Selected Range ---	-----			
0x2C9A	0x2D7	VertexShaderCode17		249	633	

Figure 3. Intel VTune Performance Analyzer disassembly mixed with shader source.

6.5 Programming the Pixel Shader

There are multiple methods for programming pixel shaders, but we highly recommend using a programming language such as HLSL, Cg, or GLSL to get the most efficient code.

There are a few things to take into account when using PS 2.0 on Intel Graphics, several of which are common to programming vertex shaders.

Reduce temp registers: The fewer temp registers used, the more likely your shader can be reorganized and instructions prioritized.

Declare coordinates in sequential groups: When you declare your texture coordinates, try to group your 2D texture lookups together before you declare any 3D coordinates. The interpolated locations are calculated four at a time, so fewer computations will be required if two can be done at once.

The following table depicts efficient texture lookups. In the first column, no passes can be combined into a group of four. In the second column, t0 and t1 can be combined in a single pass through the interpolator:

3 pass	2 pass
dcl t0.xy	dcl t0.xy
dcl t1.xyz	dcl t1.xy
dcl t2.xy	dcl t2.xyz

Minimize phases: Reducing the number of phases required to execute a shader can greatly improve performance. Try to group your dependent texture loads so they can be executed together.

Avoid unused decl's: If you declare coordinates, attributes, or components in the shader that are not used, the hardware still has to interpolate those values, wasting cycles.

Avoid writing z-value: When the shader compiler detects that you will be writing to **z** in your pixel shader, it disables intermediate **z** earlier in the pipeline, increasing unnecessary work.

Target highest instruction set: Pixel shaders written for lower PS models must be translated up to 2.0 before execution, which slows things down. By providing a PS 2.0 path, you will get the best performance possible on IGMA 950.

6.6 Debugging Pixel and Vertex Shaders

When developing shaders using DirectX, there are some easy steps for enabling shader debug. This will allow you to step through the code and set breakpoints. You must have the DirectX 9 SDK along with the shader debugger installed. See the MSDN* Shader Debugger documentation* for more information.

Switch to DirectX Debug: Though the Control Panel, select DirectX, which opens the window shown in Figure 4. Switching to the debug runtime will also enable debug output in Visual Studio*, which will allow you to catch a number of performance problems.

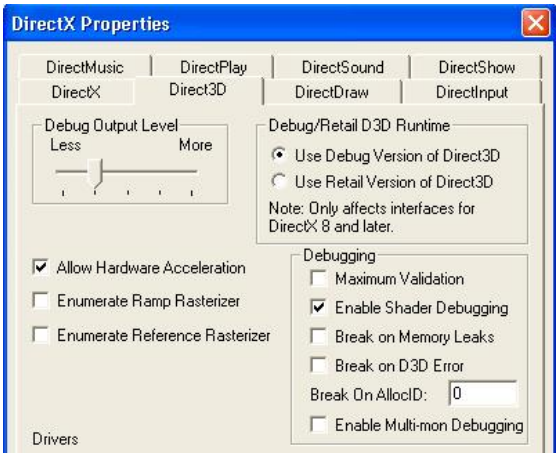


Figure 4. The DirectX Properties page.

Force software vertex and pixel shading: Software vertex processing is required to run on IGMA at all, so you probably already have that enabled with

`D3DCREATE_SOFTWARE_VERTEXPROCESSING`. For software pixel shading, you must force the device to run through the reference rasterizer using `pDeviceSettings->DeviceType=D3DDEVTYPE_REF;`

Add shader compile flags: It is a good idea to disable shader optimizations while debugging; otherwise, your source may not match what is being executed:

```
dwShaderFlags |= D3DXSHADER_DEBUG |
D3DXSHADER_SKIPOPTIMIZATION |
D3DXSHADER_FORCE_VS_SOFTWARE_NOOPT |
D3DXSHADER_FORCE_PS_SOFTWARE_NOOPT;
```

Start your application with .NET*: Once you have applied the above steps, you must start your application with Direct3D Debugging enabled, as shown in Figure 5.

Figure 5. Starting with the Direct3D Debugger.

There is limited support for debugging shaders for OpenGL. One method presented at SIGGRAPH recommends using 'printf' style debugging calls within your code[4]; writing out registers, and recompiling as necessary to narrow down issues. This technique works, but is far from optimal and an area for future work. Purcell also demonstrated Shadesmith*, an alpha tool for stepping through shaders.

There are several tools available today that allow you to visualize the shaders you are creating in real time, but you cannot step through the code, which limits what you can do. Following are a few available tools:

- FX Composer*
- RenderMonkey*
- gDEBugger*

7 Tools to Help in Programming the 915G

The Intel® Graphics Media Accelerator Profiler V2.0 (**IGMAP**) is a graphics performance profiling tool capable of real-time standalone monitoring of Intel® 900 Series Integrated Graphics hardware. It functions under 32-bit and 64-bit Intel® Extended Memory 64 Technology (Intel® **EM64T**) Windows operating systems. The tool provides full performance integration with Microsoft PerfMon*, PIX* (both Frame and Time based sampling) and Intel VTune Performance Analyzer. IGMAP is capable of monitoring both Direct3D and OpenGL applications running on Intel 900 series Integrated Graphics solutions. It is available to all registered Intel® Developer Services users.

The main goal of IGMAP is to aid developers in finding performance bottlenecks and to serve as an aid in overcoming them for Intel 900 Series integrated graphics solutions.

The tool does this by illustrating architectural bottlenecks and coding problem areas within applications that ran from its configuration submenu. The tool offers some coding advice in its help section as to possible solutions. IGMAP is a critical tool to help you broaden your applications to the integrated market.

7.1 The Intel® Graphics Media Accelerator Profiler

IGMAP interfaces with the chipset-architecture of the Intel 900 Series Integrated Graphics motherboard; through this interface, the following statistics that would normally be unavailable can be gathered[5]:

- The Number of Polygons Entering the Hardware Binner[6] per Second and Frame.
- The Number of Polygons Entering the Renderer per Second and Frame.
- The Total Pixels Rendered* per Second and Frame.
- Pixel Shader Loads per Second and Frame.

To complement the hardware information being monitored within the Intel 900 Series Integrated Graphics chipset, IGMAP also hooks and monitors the following driver (software) counters:

- Whether the program is in Hardware Zone Rendering[7].
- The Number of Total Clears* and the Ratio of Fast and Total Clears.
- The Number of Z-Evicts.
- The Number of Mid Scene Flushes (MSF) and their causes.
- The Total Video/Texture Memory.
- The number of Frames per Second (FPS).
- The Orientation of the Frame Buffer.

Zone Rendering uses a tile-based rendering system designed to reduce memory bandwidth and maximize rendering performance. It is key to optimal performance of the Intel 900 Series graphics engine. When the indicator light is Green or the “Zone Rendering” screen text in the overlay display is green, this means that Hardware Zone Rendering is active and that the Graphics engine is running in its optimal rendering state. When the indicator light is red, the engine has fallen back to classical rendering, and there may be coding issues that need to be addressed. Refer to the Profiler Help for potential performance limiters to Hardware Zone Rendering or the 915G/GM Software Developers Guide.

IGMAP closely monitors the number of z-evictions and clears (that is, **stencil** and **z**) done to the graphics hardware, either intentionally or unintentionally. It also monitors pipeline flushes done in the middle of a scene, which can adversely affect the performance of any tile-based renderer, and their causes:

- **3D Dependent Operation** – The Media and Switch Fabric (MSF) is triggered by an operation on a resource that is currently dependent on 3D commands (that is, BLT or Lock on an active 3D Texture).

- **Out Of Memory** – The MSF is triggered by an out-of-memory condition (that is, textures, vertex buffers, command buffers).
- **Set Render Target –** The MSF is triggered by swapping the current render target.
- **Set Depth Buffer** – The MSF is triggered by swapping the current depth buffer.
- **Internal** – The MSF is triggered by some other internal or an atypical event.

Under ideal conditions, these values would all be zero; however, it is often necessary to use a function or a call that causes one of the above events to occur. The best way to optimize the code is to make it so that these events do not occur when they are unwanted, and to limit the number of such "wanted" events to as few as possible.

In addition to a "Frames per Second" indicator, the Profiler also indicates Memory usage for Video and Textures in memory. Finally, the "Orientation" statistic reflects the current orientation of the frame buffer; sometimes, the framebuffer rotation is intentional, but when it is not intentional, it is a good idea to change the buffer back, because rotated modes are slower.

Figure 6. Intel ® Graphics Media Accelerator Profiler Graphic User Interface.

The statistics in Figure 6 can also be displayed in the upper left-hand corner of the screen, as shown in Figure 8, and can also be logged in a Microsoft Excel* spreadsheet format[8]. The GUI displays its data samples on a set one-second time interval so as not to

be a system performance limiter. To get frame-based samples, use the Microsoft PIX tool with the Profiler.

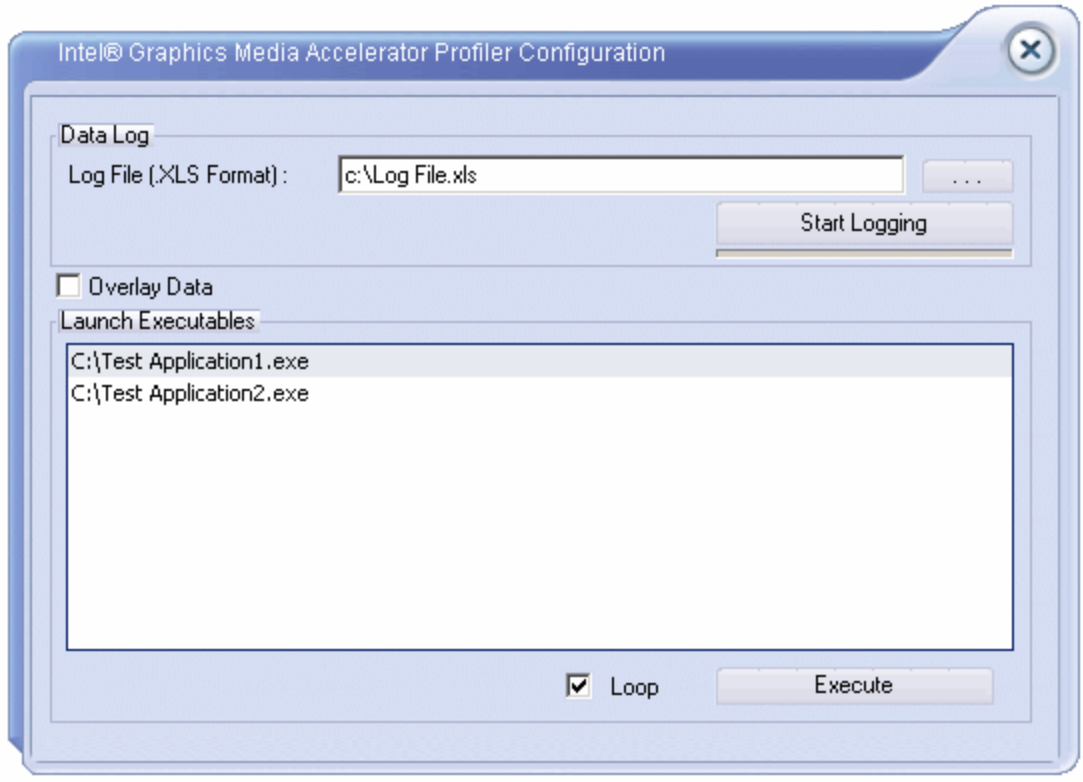


Figure 7. Intel® Graphics Media Accelerator Profiler Configuration Interface.

Once the IGMAP has been installed on a system, you will be able to track all of the above variables, down to the line of code on which they occurred, using the Intel VTune Performance Analyzer. Start the analyzer and look for the IGMAP counters in the counter-monitor wizard section. As stated earlier, the Profiler also integrates with Microsoft PIX and Microsoft PerfMon. The tool expands to allow for frame-based performance monitoring in addition to time-based sampling.

The Profiler will work on any Intel 900 Series computer (with Intel Integrated Graphics), as long as that computer has the latest Intel graphics drivers, which can be downloaded with the installation from Intel Developer Services.

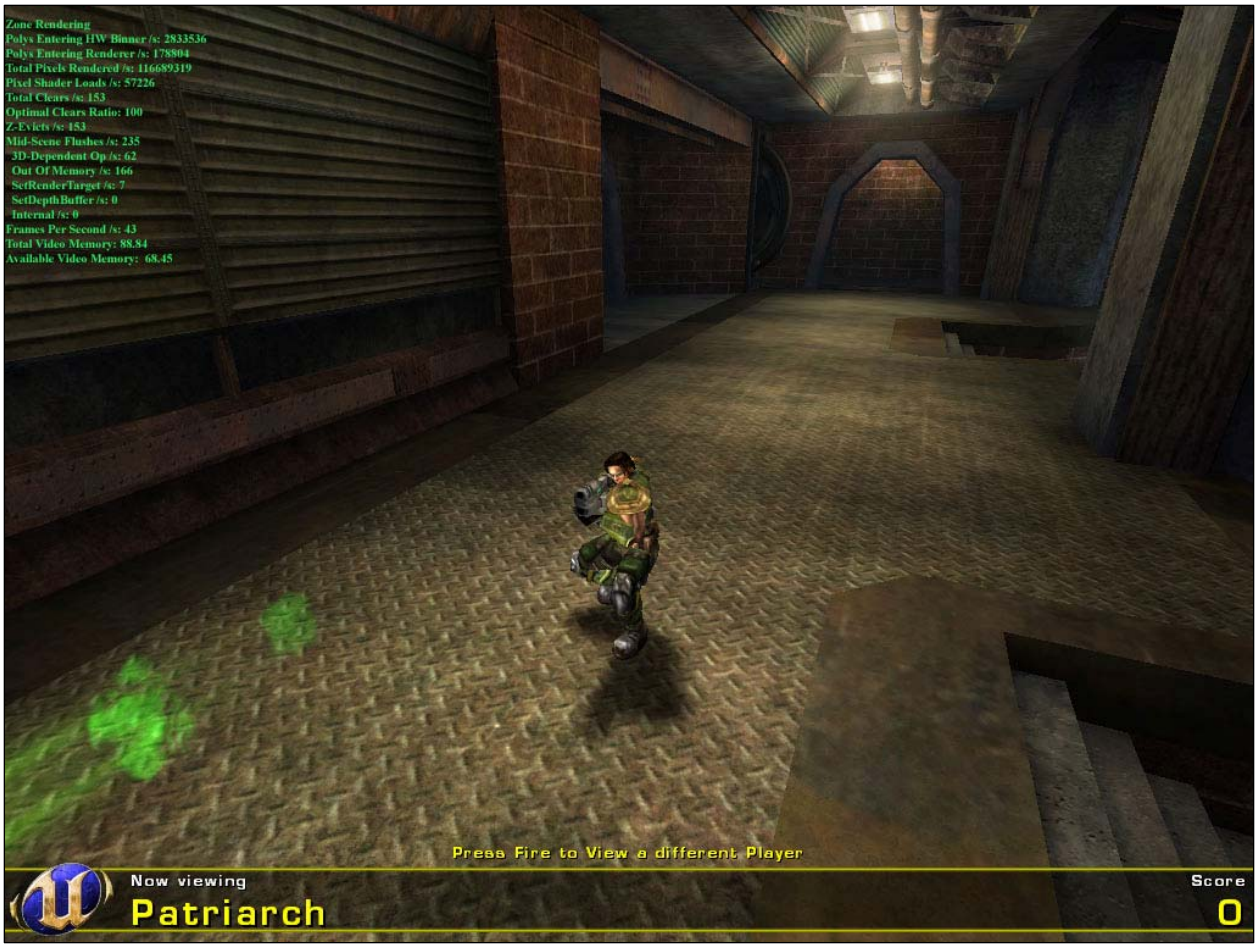


Figure 8. An Intel Graphics Media Accelerator Profiler full-screen[9] display example. Notice the information in the upper left corner of the screen.

7.2 Using the IGMAP Microsoft PIX for Windows Plug-in

Microsoft PIX for Windows has a plug-in architecture to provide counters to expose data about GPU-specific activity. This is valuable when looking into why an application is performing poorly (one example might be where the architecture is hitting a certain threshold beyond which performance drops). The IGMAP plug-in can help highlight these situations in your application.

Microsoft PIX for Windows[10] resides in the *Utilities* folder of the DirectX 9.0C SDK. This is also where PIX plug-ins (such as the 915G/GM plug-in) need to reside. Select *New Experiment* from the *File* Menu option. Use the provided *Browse* button to select the executable to be profiled. If the startup folder is not the same as the program path folder, enter it in the Startup Folder edit box. Enter the command-line parameters (if required) in the command-line *arguments* edit box.

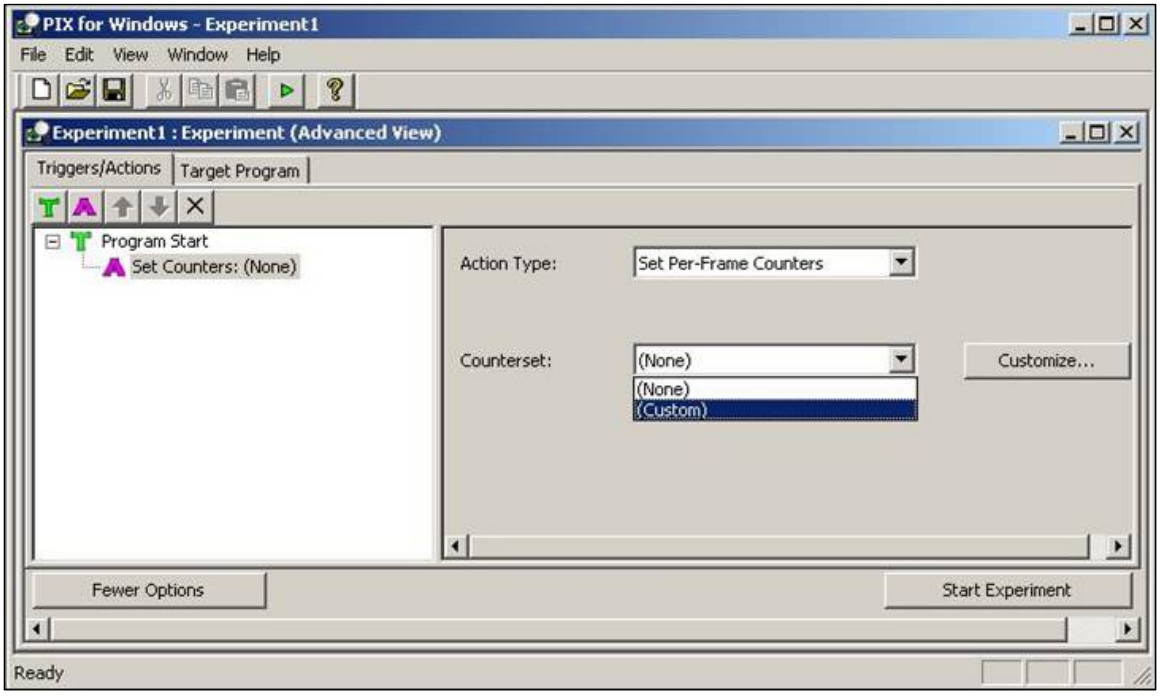


Figure 9. Selecting custom counters.

Next, select counters by clicking the *Customize* button, as shown in Figure 9. Use the *Add>>* and *Remove<<* buttons to make a list of the counters as seen in Figure 10 that will be profiled for your application. When the list is done, click *OK*. Use the Frame data gathering options to set the period during which the application will be profiled. This can be specified in both seconds and frames. Next, start the profiling session by selecting *Start Experiment*.

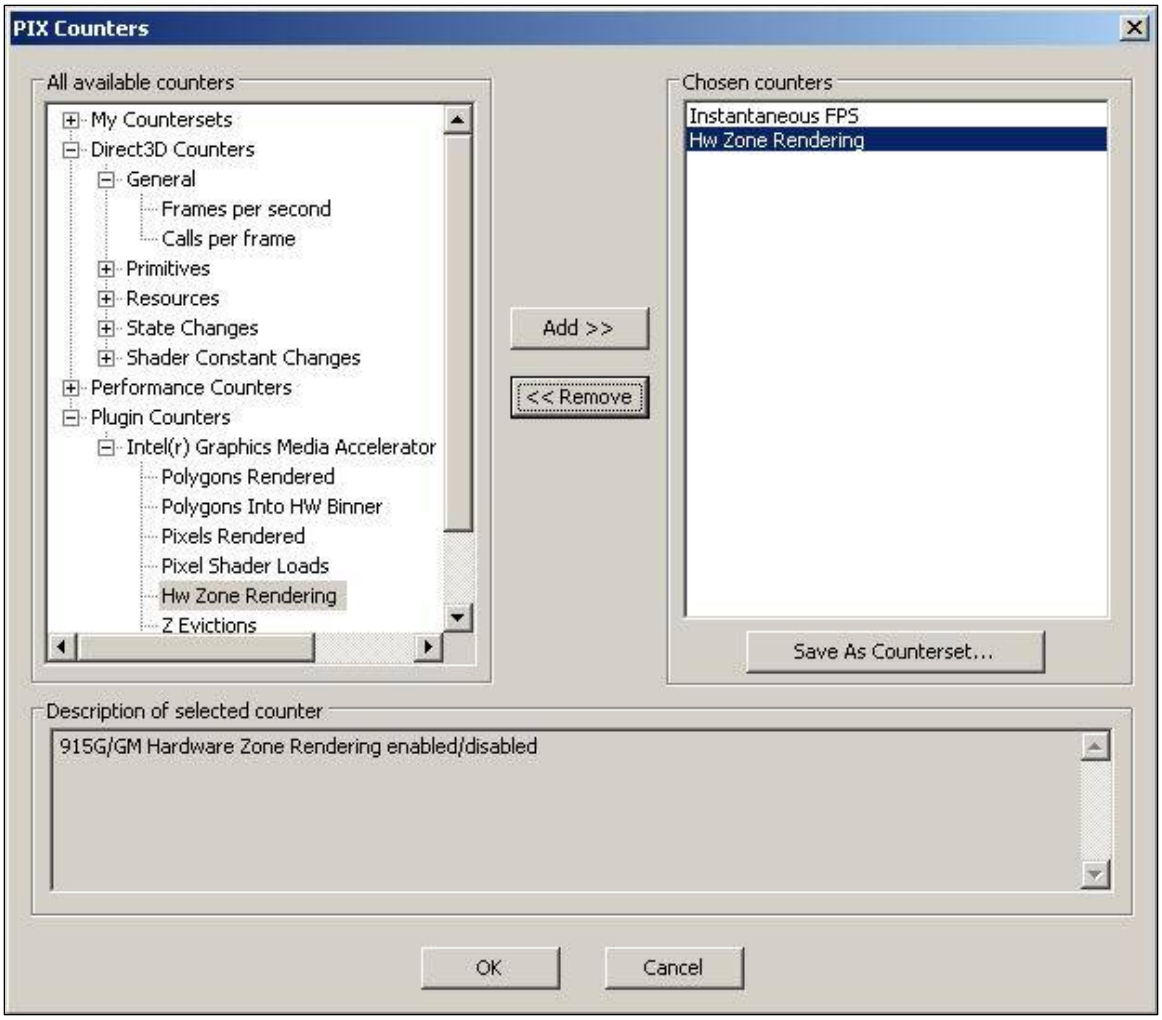


Figure 10. Selecting individual counters.

To get the results of the test that has just been run, go to *File->Open* and select the **<yoursessionname>.PIXRun** file name that has been previously specified as the output file name, as seen in Figure 11. The **Summary** Section holds general information about the conducted test such as Target Application, the DLL used, Experimental File (saved via configuration settings), and the Run File.

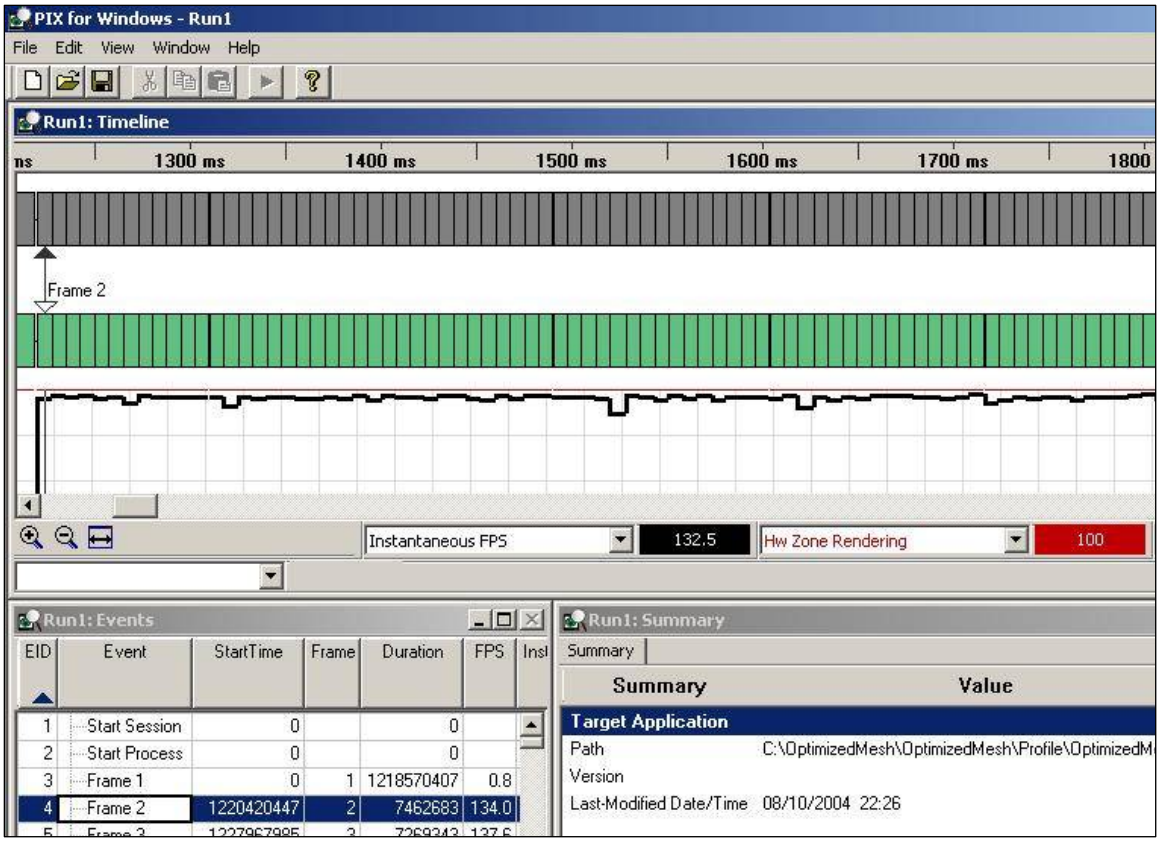


Figure 11. Example PIX output (per frame data).

In summary, the Intel Graphics Media Accelerator Profiler is a powerful tool in aiding developers to find performance bottlenecks on 900 Series Integrated Graphics hardware. The purpose of finding these bottlenecks is to provide insight not previously available when tuning for Intel 900 Series graphics hardware. The tool is available free of charge to anyone registered with Intel Developer Services.

8 How to File a Suspected Driver Bug or Feature Request

Video-miniport, display, and 3D acceleration drivers are constantly evolving, with new features and performance improvements. Because of the complexity of these drivers, bugs will crop up from time to time. Similarly, the graphics architecture team is making decisions on what features to include in the next product, and your feedback as a customer is of tremendous value.

It is important to note that a vast majority of bugs that developers believe to be driver bugs turn out to be configuration defects in their application and would occur on any hardware with similar capability. For example, if a bug appears only on Intel hardware, the problem may be an application defect that is only exposed when utilizing software vertex processing. Forcing third-party video cards to use capabilities similar to Intel devices will often expose application defects.

Should you have a feature request or a bug report to file, your first contact should be your company's main Intel Corporation contact, typically known as a Strategic Relations Manager (SRM) or Developer Relations Manager (DRM). If your company does not have one, it is very likely that your publisher does. This should be the same person you turn to for CPU information and optimizations.

Your contact is part of a team dedicated to Software Solutions for Intel products. When they have enough investigation to suspect the problem is caused by Intel hardware or software (drivers), they will work with the Intel® Desktop Platforms Group ISV Enabling team, which will debug the problem.

9 Conclusion

The rapid adoption of Intel Integrated Graphics makes this platform a worthy development and optimization target for developers. The high ratio of performance to cost have combined to make it the hardware solution of choice for a large number of mainstream users. Those users are increasingly sophisticated in their expectations of robust graphics in their games and other entertainment applications, even on modestly priced systems. ISVs who meet those expectations may enjoy a competitive advantage in the marketplace, as they succeed in improving the user experience and differentiating their products from their peers.

By taking advantage of the features of the Intel Graphics Media Accelerator (Intel GMA) 900 Series, such as Zone Rendering, Dynamic Video Memory, and Software Vertex Processing, developers can improve performance on these platforms. Particular attention is warranted to maintain zone rendering, using the guidelines covered in this guide, in order to take advantage of the performance advantages it engenders. Likewise, software vertex shading brings multiple advantages over hardware implementations; study and implementation of the techniques provided here are likely to pay off.

Tools such as the Intel VTune Performance Analyzer, the Intel GMA Profiler, and Microsoft PIX help to automate the task of obtaining high performance from Intel Integrated Graphics. With these tools and techniques, developers who optimize their applications for Intel Integrated Graphics stand to benefit by expanding their user bases.

Additional Resources

Intel Developer Services: Games Developer Center

Dean Macri. Software Vertex Processing. ShaderX2: Shader Programming Tips and Tricks with DirectX9. 2003.

Kent F. Knox. "Optimizing DirectX 9.0 Vertex Shaders for Software Vertex Processing. ShaderX3: Advanced Rendering with DirectX and OpenGL. Charles River Media. 2005.

Intel Graphics Software Developer's Guide. Revision 2.0.

Adam Lake and Cody Northrup. High Dynamic Range Environment Mapping on Mainstream Graphics Hardware. Forthcoming on gamedev.net.

Dephi3D OpenGL Hardware Registry*.

Kalra, Anu. Optimizing DirectX Applications using Software Vertex Processing. Meltdown 2003 Presentation.

Purcell, Tim. Fragment Program Debugging Tools*, SIGGRAPH 2004.

Contributors and Reviewers

Mike Apodaca, Ben Ashbaugh, Chuck Desylva, Sabina Khasat, Adam Lake, Dean Macri, Cody Northrop, Kipp Owens[11], Teb Pabon, Kim Pallister, Katen Shah, Chris Silva, Ronen Zohar

---

[1] Source: Mercury Research PC Graphics Report Q32004

[2] Ronen Zohar, Kim Pallister. Optimizing DirectX Applications using Software Vertex Processing*. Meltdown 2001 Presentation. 2001.

[3] Kalra, Anu. Optimizing DirectX Applications using Software Vertex Processing. Meltdown 2003 Presentation.

[4] Purcell, Tim. Fragment Program Debugging Tools*, SIGGRAPH 2004.

[5] See previous sections for detailed description of these terms (i.e. HW Binning, HW Zone Rendering, etc).

[6] The HW Binner is a core HW component to the Intel® Tile Based rendering engine.

[7] Note: Falling out of HW Zone Rendering is one of the largest performance limiters of Intel® 900 Series Integrated Graphics Hardware

[8] Note: The current version logs only time-based data (not Frame-based) samples.

[9] Picture comes from Unreal Tournament 2004 Demo. Unreal Tournament is proprietary of Epic Games
Incorporated.

[10] Introduced in the summer of 2004.

[11] Kipp Owens was the author of the 2004 version of the Intel Graphics Developer's Guide. We have retained significant portions of his work in this updated version.

Appendix: Creating a DirectX 9 Device and Identifying Intel Graphics

The program below is a very simple one that checks the device capabilities of the primary adapter and sets the vertex processing mode to software transform and lighting if it detects an Intel Graphics chipset. A Direct3D device is created and a single triangle is drawn.

The program is a modification of the DirectX 9 SDK tutorial "Vertices." It requires the DirectX 9 SDK to compile and must be linked with d3d9.lib. This is adapted from Intel® Graphics Software Developer's Guide, Revision 2.0.

```
#include <d3d9.h>
#include <string.h>

//-----------------------------------------------------------------------
-------
// Global variables
//-----------------------------------------------------------------------
-------
LPDIRECT3D9             g_pD3D       = NULL; // Used to create the
D3DDevice
LPDIRECT3DDEVICE9       g_pd3dDevice = NULL; // Our rendering device
LPDIRECT3DVERTEXBUFFER9 g_pVB        = NULL; // Buffer to hold vertices
DWORD    g_VertexProcessingMode = 0; // Used to set SW or HW vert proc.

// A structure for our custom vertex type
struct CUSTOMVERTEX
{
    FLOAT x, y, z, rhw; // The transformed position for the vertex
    DWORD color;        // The vertex color
};

// Our custom FVF, which describes our custom vertex structure
#define D3DFVF_CUSTOMVERTEX (D3DFVF_XYZRHW|D3DFVF_DIFFUSE)
#define E_MINSPEC (-3)  // Error code for parts not meeting min spec
```

```
//-----------------------------------------------------------------------
-------
// Name: SetVertexProcessingMode
// Desc: Checks HW TnL caps and IDs 865G to enable SW TnL
//-----------------------------------------------------------------------
-------
DWORD SetVertexProcessingMode( LPDIRECT3D9 pD3D )
{
   DWORD       vertexprocessingmode;  // vertex processing mode
   D3DCAPS9    caps;                  // structure that stores device
caps...
   D3DADAPTER_IDENTIFIER9 adapterID;  // Used to store device info

       // Check the capabilities...store into "caps"...
   if( g_pD3D->GetDeviceCaps( 0, D3DDEVTYPE_HAL, &caps ) != D3D_OK )
   {
         return E_FAIL; // exit if reading caps fails...
   }

   // check if hardware TnL is supported...
   if ( ( caps.DevCaps & D3DDEVCAPS_HWTRANSFORMANDLIGHT ) != 0 )
   {
         vertexprocessingmode = D3DCREATE_HARDWARE_VERTEXPROCESSING;
   }
   else
   {
         // check vendor and device ID and enable software vertex
processing
         // for Intel(R) Graphics...
```

```
        // Gather the primary adapter's information...
        if( g_pD3D->GetAdapterIdentifier( 0, 0, &adapterID ) !=
D3D_OK )
        {
              return E_FAIL;
        }


             if ( ( adapterID.VendorId == 0x8086 ) &&
( ( adapterID.DeviceId == 0x2582 ) ||
( adapterID.DeviceId == 0x2782 ) ||
( adapterID.DeviceId == 0x2572 ) ||
( adapterID.DeviceId == 0x3582 ) ||
( adapterID.DeviceId == 0x2562 ) ||
( adapterID.DeviceId == 0x3577 ) ) )
                                                    {
              vertexprocessingmode =
D3DCREATE_SOFTWARE_VERTEXPROCESSING;
        }
        else
        {
              // chip does not meet requirements...
              return E_MINSPEC;
        }
   }

   return vertexprocessingmode;
}
```

```
//-----------------------------------------------------------------------
-------
// Name: InitD3D()
// Desc: Initializes Direct3D
//-----------------------------------------------------------------------
-------
HRESULT InitD3D( HWND hWnd )
{
    // Create the D3D object.
    if( NULL == ( g_pD3D = Direct3DCreate9( D3D_SDK_VERSION ) ) )
        return E_FAIL;

    char mode_str[255];
     g_VertexProcessingMode = SetVertexProcessingMode( g_pD3D );

    // Set up the structure used to create the D3DDevice
    D3DPRESENT_PARAMETERS d3dpp;
    ZeroMemory( &d3dpp, sizeof(d3dpp) );
    d3dpp.Windowed = TRUE;
    d3dpp.SwapEffect = D3DSWAPEFFECT_DISCARD;
    d3dpp.BackBufferFormat = D3DFMT_UNKNOWN;

    switch( g_VertexProcessingMode )
    {
   case E_FAIL:     MessageBox( hWnd, "Error identifying GPU", "Error",
MB_OK );
                          exit( E_FAIL );

   case E_MINSPEC: MessageBox( hWnd, "GPU does not meet minimum specs:
Intel(R) Graphics or Hardware T&L chip required", "Error", MB_OK );
                          exit( E_MINSPEC );

   case D3DCREATE_HARDWARE_VERTEXPROCESSING:
                          strcpy( mode_str, "Hardware T&L Enabled" );
```

```
                        break;
   case D3DCREATE_SOFTWARE_VERTEXPROCESSING:
                        strcpy( mode_str, "Software T&L Enabled" );
                        break;
    }

    if( FAILED( g_pD3D->CreateDevice( D3DADAPTER_DEFAULT, D3DDEVTYPE_HAL,
hWnd,
                                g_VertexProcessingMode, &d3dpp,
&g_pd3dDevice ) ) )
    {
   return E_FAIL;
    }

    // Device state would normally be set here
    MessageBox( hWnd, mode_str, "Vertex Processing Mode", MB_OK );
    return S_OK;
}




//-----------------------------------------------------------------------
-------
// Name: InitVB()
// Desc: Creates a vertex buffer and fills it with our vertices. The
vertex
//       buffer is basically just a chuck of memory that holds vertices.
After
//       creating it, we must Lock()/Unlock() it to fill it. For indices,
D3D
//       also uses index buffers. The special thing about vertex and
index
//       buffers is that they can be created in device memory, allowing
some
```

```
//       cards to process them in hardware, resulting in a dramatic
//       performance gain.
//-----------------------------------------------------------------------
-------
HRESULT InitVB()
{
    // Initialize three vertices for rendering a triangle
    CUSTOMVERTEX vertices[] =
    {
        { 150.0f,  50.0f, 0.5f, 1.0f, 0xffff0000, }, // x, y, z, rhw,
color
        { 250.0f, 250.0f, 0.5f, 1.0f, 0xff00ff00, },
        {  50.0f, 250.0f, 0.5f, 1.0f, 0xff00ffff, },
    };

    // Create the vertex buffer. Here we are allocating enough memory
    // (from the default pool) to hold all our 3 custom vertices. We
also
    // specify the FVF, so the vertex buffer knows what data it contains.
    if( FAILED( g_pd3dDevice->CreateVertexBuffer( 3*sizeof(CUSTOMVERTEX),
                                                  g_VertexProcessingMode,
                                                   D3DFVF_CUSTOMVERTEX,
                                                   D3DPOOL_DEFAULT,
&g_pVB, NULL ) ) )
    {
        return E_FAIL;
    }

    // Now we fill the vertex buffer. To do this, we need to Lock() the
VB to
    // gain access to the vertices. This mechanism is required becuase
vertex
    // buffers may be in device memory.
    VOID* pVertices;
```

```
    if( FAILED( g_pVB->Lock( 0, sizeof(vertices), (void**)&pVertices,
0 ) ) )
        return E_FAIL;
    memcpy( pVertices, vertices, sizeof(vertices) );
    g_pVB->Unlock();

    return S_OK;
}




//-----------------------------------------------------------------------
-------
// Name: Cleanup()
// Desc: Releases all previously initialized objects
//-----------------------------------------------------------------------
-------
VOID Cleanup()
{
    if( g_pVB != NULL )
        g_pVB->Release();

    if( g_pd3dDevice != NULL )
        g_pd3dDevice->Release();

    if( g_pD3D != NULL )
        g_pD3D->Release();
}




//-----------------------------------------------------------------------
-------
```

```
// Name: Render()
// Desc: Draws the scene
//-----------------------------------------------------------------------
-------
VOID Render()
{
    // Clear the backbuffer to a blue color
    g_pd3dDevice->Clear( 0, NULL, D3DCLEAR_TARGET, D3DCOLOR_XRGB(0,0,0),
1.0f, 0 );

    // Begin the scene
    if( SUCCEEDED( g_pd3dDevice->BeginScene() ) )
    {
        // Draw the triangles in the vertex buffer. This is broken into
a few
        // steps. We are passing the vertices down a "stream", so first
we need
        // to specify the source of that stream, which is our vertex
buffer. Then
        // we need to let D3D know what vertex shader to use. Full,
custom vertex
        // shaders are an advanced topic, but in most cases the vertex
shader is
        // just the FVF, so that D3D knows what type of vertices we are
dealing
        // with. Finally, we call DrawPrimitive() which does the actual
rendering
        // of our geometry (in this case, just one triangle).
        g_pd3dDevice->SetStreamSource( 0, g_pVB, 0,
sizeof(CUSTOMVERTEX) );
        g_pd3dDevice->SetFVF( D3DFVF_CUSTOMVERTEX );
        g_pd3dDevice->DrawPrimitive( D3DPT_TRIANGLELIST, 0, 1 );

        // End the scene
        g_pd3dDevice->EndScene();
```

```
    }

    // Present the backbuffer contents to the display
    g_pd3dDevice->Present( NULL, NULL, NULL, NULL );
}




//-----------------------------------------------------------------------
-------
// Name: MsgProc()
// Desc: The window's message handler
//-----------------------------------------------------------------------
-------
LRESULT WINAPI MsgProc( HWND hWnd, UINT msg, WPARAM wParam, LPARAM
lParam )
{
    switch( msg )
    {
        case WM_DESTROY:
            Cleanup();
            PostQuitMessage( 0 );
            return 0;
    }

    return DefWindowProc( hWnd, msg, wParam, lParam );
}




//-----------------------------------------------------------------------
-------
// Name: WinMain()
```

```
// Desc: The application's entry point
//-----------------------------------------------------------------------
-------
INT WINAPI WinMain( HINSTANCE hInst, HINSTANCE, LPSTR, INT )
{
    // Register the window class
    WNDCLASSEX wc = { sizeof(WNDCLASSEX), CS_CLASSDC, MsgProc, 0L, 0L,
                      GetModuleHandle(NULL), NULL, NULL, NULL, NULL,
                      "Enabling Software T&L for 865G", NULL };
    RegisterClassEx( &wc );

    // Create the application's window
    HWND hWnd = CreateWindow( "Enabling Software T&L for 865G",
                        "Intel(R) 865G Detection/Initialization",
                      WS_OVERLAPPEDWINDOW, 100, 100, 300, 300,
                              GetDesktopWindow(), NULL, wc.hInstance,
NULL );

    // Initialize Direct3D
    if( SUCCEEDED( InitD3D( hWnd ) ) )
    {
        // Create the vertex buffer
        if( SUCCEEDED( InitVB() ) )
        {
            // Show the window
            ShowWindow( hWnd, SW_SHOWDEFAULT );
            UpdateWindow( hWnd );

            // Enter the message loop
            MSG msg;
            ZeroMemory( &msg, sizeof(msg) );
            while( msg.message!=WM_QUIT )
            {
```

```
                if( PeekMessage( &msg, NULL, 0U, 0U, PM_REMOVE ) )
                {
                    TranslateMessage( &msg );
                    DispatchMessage( &msg );
                }
                else
                    Render();
            }
        }
    }

    UnregisterClass( "Enabling Software T&L for 865G", wc.hInstance );
    return 0;
}
```